Self-Testing Fault-Tolerant Real-Time Systems

M. Rooholamini and S. H. Hosseini

Electrical Engineering and Computer Science Department
University of Wisconsin - Milwaukee
Milwaukee, WI 53201, USA
Email: {mr,hosseini}@cs.uwm.edu

Abstract. We propose a periodic diagnostic algorithm based on the testing model of computation for real-time systems. The diagnostic task runs on every processor of the system. When the task starts execution, all the processors are synchronized and will be doing the same operation at every step of the algorithm. Each processor performs a test of itself and generates a token which contains the test result. Then the token is passed to some neighboring processors to check if a failure has occurred. In our model, a faulty processor does not necessarily stop functioning and it may behave in erratic manners when checking the token of a processor it is assigned to test. We give the conditions under which all processor failures are detected in a torus interconnection network, where each processor is tested by a minimum number of processors.

1 Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical correctness of the computation, but also on the time at which the results are produced [9]. In such systems, each task has a deadline by which it must complete execution. More and more real-time systems are relying on multiprocessors [11]. Multiprocessor systems, in general, are more reliable than a single processor system, because in these systems failure of one processor does not necessarily cause the whole system to fail if some fault-tolerant techniques are provided [7]. However, in many real-time research studies, multiprocessor scheduling is done either without concern for fault-tolerance, [2, 10, 12], or it is assumed that the hardware support or additional mechanisms to detect the failures is provided and the problem of detecting the faults in the first place is ignored [1, 7, 13]. In addition, most of these works assume a *fail-stop* model, where as soon as a failure occurs the processor stops functioning.

In this paper, we propose a periodic real-time diagnostic algorithm for two-dimensional torus interconnection networks (ICN) [5], where the processors are responsible for running application tasks as well as the diagnostic task. The diagnostic task is based on the testing model of computation [4, 6]. We assign a processor to test itself and the result of the test will be checked by some neighboring processors to determine if a processor failure has occurred. Our model is as opposed to the comparison model of computation [3, 14], in which each task is assigned to at least two processors and the outputs produced are

compared to determine if a failure has occurred. The comparison model has the disadvantage of reducing the system's throughput, as two or more processors are assigned to execute the same tasks. Furthermore, we assume a more realistic model, *fail-continue* model, where a failed processor does not necessarily stop functioning and it may behave in erratic manners. In a related work [8], the author assumes a system of two processors, which communicate with each other by passing messages, and determines an upper bound on the time a message should be received by each processor. In that work, under the fail-stop model and the assumption that at least one of the two processors is fault-free, the faulty processor is detected, if a message is not received by the given bound.

In the next section, we describe our proposed model and give the diagnostic algorithm.

2 The Proposed Model

First, we state our assumptions:

1. There is a fault-free host supervising a torus ICN whose processors are subject to failure. The processors of ICN execute application tasks and are responsible for running the diagnostic task and checking the test result of their neighbors.
2. Every processor has an I/O port connected to it.
3. The communication network is fault-free.
4. A fault-free processor correctly diagnoses a processor it tests.
5. The test of a processor performed by a faulty processor produces an unpredictable result, if any. A faulty processor may diagnose a fault-free (faulty) processor as fault-free or faulty.
6. No processor failure occurs during a run of the diagnostic program.

2.1 The Diagnostic Program

The diagnostic program DP is a periodic task that is characterized by a 2-tuple (C, P), where C is its computation time and P is its period. In real-time systems, any fault-detection procedure employed must be predictable and time-bounded. An instance of DP starts execution on every processor of a torus at time tP, for $t = 0, 1, 2, \ldots$, and runs for C time units, by the end of which faulty processors are detected. This is done by prescheduling every processor as in Figure 1. The time intervals $[0, C], [P, P + C], [2P, 2P + C], [3P, 3P + C], \ldots$, are reserved on each processor for DP. The application tasks which can be either periodic or aperiodic will be scheduled in time intervals $[C, P], [P + C, 2P], [2P + C, 3P], [3P + C, 4P], \ldots$, based on preemptive, non-preemptive, priority driven, or any scheduling policy such as those proposed in [2, 12, 10]. The computation time C of DP, which must be bounded, can be determined for an underlying topology. This parameter is a function of communication cost and speed of a processor. When DP starts execution, all the processors are synchronized and

will be performing the same task at every clock tick. If two processors need to communicate at some step of DP, the link connecting them will be dedicated to only those two processors. As a result, C is computed as the sum of the total execution time of instructions of DP, on any processor, and the total communication time between any two adjacent processors. P can be selected as a design parameter. There is a trade-off in choosing a value for P. By making P small, failures can be detected sooner after they occur and hence less application tasks miss their deadlines due to processor failure. However, this tends to reduce the system's throughput, since more processor time is spent executing instances of DP. On the other hand, making P large will result in failures to go undetected for a longer time and hence more deadlines will be missed, but it has the advantage of having more processor times allocated to application tasks.

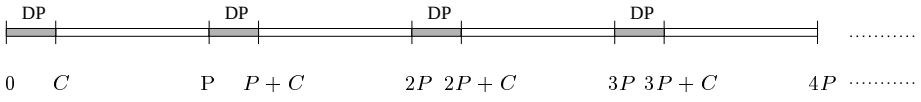

Fig. 1. Schedule of each processor for the diagnostic program, DP.

2.2 The Diagnostic Algorithm

The diagnostic algorithm DA is token based. The algorithm is made up of three phases as follows:

Phase1 When DA starts execution on a processor p_i, p_i performs a test of itself. A test step can be as simple as writing to a register and reading it back or it can involve a complex computation. After a test procedure completes, a token is generated by p_i which contains the result of the test of itself it performed.

Phase2 Processor p_i sends the generated token to its neighbors that are below it and to its left.

Phase3 Processor p_j checks the token it received from p_i to see if p_i has failed. It then reports to the host the test result. The host will make the final decision on the faulty and fault-free processors (note that, since p_j itself may be faulty, its report may not be reliable).

A 4×4 torus and its testing graph are given in Figure 2.

Next, we define the following parameters.

1. N, is the number of processors in a two-dimensional torus ICN.
2. r, is the number of processors along each dimension of ICN.

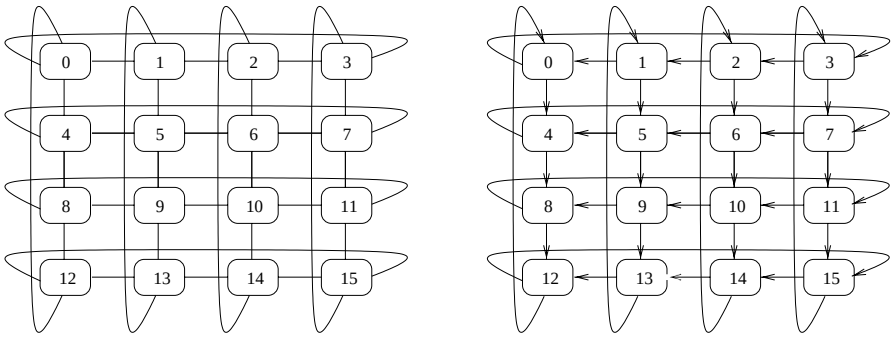

Fig. 2. A 4×4 torus and its testing graph.

3. $token_i$, is the token generated by processor p_i after performing a self-test.

4. stc, is the self-test cost (in units of time) of a processor. This is the cost of generating a token by the processor.

5. ttd, is the token transmission delay (in units of time) between two adjacent processors.

6. tcc, is the token-check cost incurred by a processor to check the token of a processor it is assigned to test.

7. $test_{p_i}(p_k, tr)$, is a message sent from processor p_i to the host after it performs a test of $token_k$. If p_i has concluded that p_k is faulty, test result $tr = 1$. Otherwise, $tr = 0$.

8. mtd, is the message transmission delay incurred when p_i sends a $test_{p_i}()$ message to the host, at the end of a round of DA.

9. $FV[tr_1, tr_2, \ldots, tr_N]$, is the fault vector that continuously gets updated by the host after a round of testing by the processors. If the host concludes that processor p_i is faulty, then $tr_i = 1$, otherwise, $tr_i = 0$.

10. p_i^d, is the processor at location d with respect to processor p_i, for $d \in \{n, s, e, w\}$. These are the processors that are to the north, south, east, and west of p_i (i.e., all neighbors of p_i).

11. $F(p_i, d)$, is a function that maps to processor p_i^d, with respect to p_i, for

$d \in \{n, s, e, w\}$. These processors for p_i are determined as follows:

$$p_i^n = F(p_i, n) = \begin{cases} p_{i+r(r-1)}, & \text{if } 0 \leq i < r \\ p_{i-r}, & \text{otherwise} \end{cases}$$

$$p_i^s = F(p_i, s) = \begin{cases} p_{i-r(r-1)}, & \text{if } N - r \leq i < N \\ p_{i+r}, & \text{otherwise} \end{cases}$$

$$p_i^e = F(p_i, e) = \begin{cases} p_{i-(r-1)}, & \text{if } i = kr - 1, \text{ for } 0 < k \leq r \\ p_{i+1}, & \text{otherwise} \end{cases}$$

$$p_i^w = F(p_i, w) = \begin{cases} p_{i+(r-1)}, & \text{if } i = kr, \text{ for } 0 \leq k < r \\ p_{i-1}, & \text{otherwise} \end{cases}$$

Phase1 of DA starts at time tP, for $t = 0, 1, 2, \ldots$, and completes at time $(tP + stc)$, when $token_i$ is generated by processor p_i, for $0 \leq i < N$, and when Phase2 starts. At the start of Phase2, processor p_i sends $token_i$ to its testers p_i^s and p_i^w, in parallel. These are the processors that are below and to the left of p_i, respectively. At time $(tP + stc + ttd)$ Phase2 of DA terminates and Phase3 starts. If by the start of Phase3 p_i^s and/or p_i^w have not received $token_i$, they conclude that p_i is faulty. At the end of time $(tP + stc + ttd + 2tcc)$ every processor has determined if a processor it tested is faulty or fault-free. If a tester processor p_j has found a processor p_i it tested faulty, it sends the message $test_{p_j}(p_i, 1)$ to the host. Otherwise, it sends the message $test_{p_j}(p_i, 0)$. This will add an additional mtd time units to the execution time of DA, for each message sent by a tester processor. If the host does not receive a test message from a tester processor by the end of time $(tP + stc + ttd + 2tcc + 2mtd)$, it concludes that the tester processor is faulty. At this time, the host will make the final decision on status of every processor, based on the test messages it has received from the processors of ICN, and updates the fault vector $FV[]$. Any additional time this step takes is added to the computation time of DA. Once, status of each processor is determined, Phase3 of the run of DA, and hence DA, ends.

Next, we state the conditions under which all the processor failures in the network are correctly detected.

Theorem 2.1 *In a two-dimensional torus with r processors along each dimension, given that each processor is tested by two of its neighbors with at most one of them faulty. The maximum number of processors that can be faulty is $\lfloor \frac{r}{2} \rfloor \times r$.*

Proof. One way to generate the maximum number of faults is to make all the processors faulty on the first row. This makes p_i and its left tester, p_i^w, for $0 \leq i < r$, faulty. None of these processors can have its second tester, p_i^s, below it, faulty. Thus, all processors p_j, for $r \leq j < 2r$, on the second row must

be fault-free. Processors p_j on the second row have their left tester p_j^w fault-free, therefore they can have their second tester p_j^s faulty, which will make all the processors on the third row faulty. Continuing in the same fashion all the processors on every other row are faulty and all the processors on the other rows are fault-free. If r is odd, then none of the processors on the last row (row r) can be faulty. Otherwise, processors on the last row will have both their testers faulty, since the processors on row 1 are testers of the processors on row r. As a result, the maximum number of processor failures is $\lfloor \frac{r}{2} \rfloor \times r$.

Corollary 2.1 *Given that the number of processor faults in a two-dimensional torus with r processors along each dimension, employing the diagnostic algorithm, is $\left(\lfloor \frac{r}{2} \rfloor \times r\right) - 1$ or less. There must exist at least two processors with both their testers fault-free.*

Proof. In a torus with maximum number of faults, $\lfloor \frac{r}{2} \rfloor \times r$, each processor has exactly one of its testers faulty and the other one fault-free (that is, if r is even; if r is odd, there are some processors that have both their testers fault-free). Making any one of the faulty processors, say p_i, fault-free will make the processors p_i tests to have both their testers fault-free.

Theorem 2.2 *Given the diagnostic algorithm. For a two-dimensional torus with r processors along each dimension, every processor failure is correctly detected provided that there are at most $\left(\lfloor \frac{r}{2} \rfloor \times r\right) - 1$ faulty processors and each processor has at least one of its testers fault-free.*

Proof. Since, by Corollary 2.1, there are at least two processors with both their testers fault-free, there will be a processor p_k that is correctly diagnosed as faulty or fault-free, immediately. That is, for p_i and p_j testing p_k, we have

1. $test_{p_i}(p_k, 0)$ and $test_{p_j}(p_k, 0)$ or
2. $test_{p_i}(p_k, 1)$ and $test_{p_j}(p_k, 1)$.

Any such processor as p_k is correctly diagnosed as fault-free or faulty. The question that arises here is that how if p_k is a processor with one of its testers faulty which arbitrarily marks p_k as fault-free or faulty. Notice that, from the two cases above, 1 and 2, we cannot draw any conclusions on status of the testers, p_i and p_j. However, the status of p_k can be correctly diagnosed, even if one its testers has arbitrarily diagnosed it as fault-free or faulty, in which case, the faulty processor has happened to agree with the fault-free processor on the test outcome (no matter what it is).

Now, if two tester processors p_i and p_j have disagreed on a test outcome of a processor p_k, the status of p_k is not immediately known. However, it is known that either p_i or p_j must be faulty. To determine which one of them is faulty, starting from a processor whose both testers diagnosed it as either fault-free or faulty (there are at least two such processors) we can determine, in an iterative manner, all the fault-free and faulty processors. For example, if p_k is determined to be fault-free and it has disagreed with the two other testers

on the test outcomes of the processors it tested, it is concluded that the other two testers are faulty. Moreover, the statuses of the processors p_k has tested are determined once status of p_k is known. Continuing in the same fashion status of every processor is determined.

Notice that, in this algorithm the number of testers, two, each processor is tested by is minimum. If a processor is tested by only one processor, there is no way to know if a tester processor is behaving erratically, under the fail-continue model.

Example: Consider the 4×4 torus of Figure 3 with faulty processors 2, 3, 5, 6, 8, 9, and 12 grayed out for which the conditions of Theorem 2.2 hold. After a round of testing, the test results obtained for each processor are as follows, where it is assumed that processors 2, 3, 5, and 6 always misdiagnose the processors they test (i.e., they diagnose faulty processors as fault-free and vice versa), processor 8 always diagnoses the processors it tests as faulty, and processors 9 and 12 always diagnose the processors they test as fault-free.

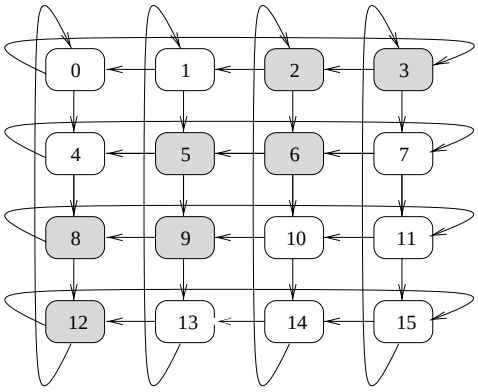

Fig. 3. A 4×4 torus with faulty processors grayed out.

$$a : \begin{cases} test_3(0,1) \\ test_4(0,0) \end{cases} \qquad b : \begin{cases} test_0(1,0) \\ test_5(1,1) \end{cases}$$

$$c : \begin{cases} test_1(2,1) \\ test_6(2,0) \end{cases} \qquad d : \begin{cases} test_2(3,0) \\ test_7(3,1) \end{cases}$$

$$e : \begin{cases} test_7(4,0) \\ test_8(4,1) \end{cases} \qquad f : \begin{cases} test_4(5,1) \\ test_9(5,0) \end{cases}$$

$$g: \begin{cases} test_5(6,0) \\ test_{10}(6,1) \end{cases} \qquad h: \begin{cases} test_6(7,1) \\ test_{11}(7,0) \end{cases}$$

$$i: \begin{cases} test_{11}(8,1) \\ test_{12}(8,0) \end{cases} \qquad j: \begin{cases} test_8(9,1) \\ test_{13}(9,1) \end{cases}$$

$$k: \begin{cases} test_9(10,0) \\ test_{14}(10,0) \end{cases} \qquad l: \begin{cases} test_{10}(11,0) \\ test_{15}(11,0) \end{cases}$$

$$m: \begin{cases} test_{15}(12,1) \\ test_0(12,1) \end{cases} \qquad n: \begin{cases} test_{12}(13,0) \\ test_1(13,0) \end{cases}$$

$$o: \begin{cases} test_{13}(14,0) \\ test_2(14,1) \end{cases} \qquad p: \begin{cases} test_{14}(15,0) \\ test_3(15,1) \end{cases}$$

From cases j, k, l, m, and n the following fault vector is immediately obtained.

$$FV = [-,-,-,-,-,-,-,-,-,1,0,0,1,0,-,-],$$

where so far it is known that processors 10, 11, and 13 are fault-free and processors 9 and 12 are faulty, as each of them has been diagnosed the same by both its testers. Processors 11 and 12 have both their testers fault-free and processors 9, 10, and 13 have only one of the testers fault-free. Even though, for example, the tester processor 8 may have arbitrarily diagnosed processor 9 as faulty, since the diagnosis matches that of the other (fault-free) tester, 13, it is concluded that processor 9 is faulty. Similarly, processor 9 may have arbitrarily diagnosed processor 10 as fault-free, which happens to match the diagnosis of 14. The same is true for processor 12, a tester of 13.

Using $FV[\,]$, we can draw conclusions on status of more processors as follows. Here, we have shown the results of previously known diagnoses only if new conclusions can be drawn from them.

9 is faulty	$\overset{\text{by } f}{\Longrightarrow}$	$test_9(5,0)$ is a misdiagnosis	$\Rightarrow$	5 is faulty
		$test_4(5,1)$ is a correct diagnosis	$\Rightarrow$	4 is fault-free
10 is fault-free	$\overset{\text{by } g}{\Longrightarrow}$	$test_{10}(6,1)$ is a correct diagnosis	$\Rightarrow$	6 is faulty
11 is fault-free	$\overset{\text{by } h}{\Longrightarrow}$	$test_{11}(7,0)$ is a correct diagnosis	$\Rightarrow$	7 is fault-free
	$\overset{\text{by } i}{\Longrightarrow}$	$test_{11}(8,1)$ is a correct diagnosis	$\Rightarrow$	8 is faulty
13 is fault-free	$\overset{\text{by } o}{\Longrightarrow}$	$test_{13}(14,0)$ is a correct diagnosis	$\Rightarrow$	14 is fault-free
		$test_2(14,1)$ is a misdiagnosis	$\Rightarrow$	2 is faulty

The updated $FV[\,]$ becomes:

$$FV = [-,-,1,-,0,1,1,0,1,1,0,0,1,0,0,-]$$

Using the new diagnoses, we continue:

2 is faulty	$\overset{\text{by } d}{\Longrightarrow}$	$test_2(3,0)$ is a misdiagnosis	$\Rightarrow$	3 is faulty
4 is fault-free	$\overset{\text{by } a}{\Longrightarrow}$	$test_4(0,0)$ is a correct diagnosis	$\Rightarrow$	0 is fault-free
5 is faulty	$\overset{\text{by } b}{\Longrightarrow}$	$test_5(1,1)$ is a misdiagnosis	$\Rightarrow$	1 is fault-free
14 is fault-free	$\overset{\text{by } p}{\Longrightarrow}$	$test_{14}(15,0)$ is a correct diagnosis	$\Rightarrow$	15 is fault-free

The updated and final $FV[\,]$ becomes:

$$FV = [0,0,1,1,0,1,1,0,1,1,0,0,1,0,0,0],$$

where all the faulty and fault-free processors are correctly diagnosed.

In what follows, we give a formal description of the algorithm:

The Diagnostic Algorithm

For $t = 0, 1, 2, \ldots,$ do

Phase1

1. Start the diagnostic program at time tP on processor p_i, for $0 \leq i < N$.
2. Processor p_i performs a self-test for stc time units and generates $token_i$.

Phase2

1. Processor p_i sends $token_i$ to p_i^w and to p_i^s at time $(tP + stc)$.
2. If p_i^w does not receive $token_i$ by time $(tP+stc+ttd)$, it generates $test_{p_i^w}(p_i, 1)$.
3. If p_i^s does not receive $token_i$ by time $(tP+stc+ttd)$, it generates $test_{p_i^s}(p_i, 1)$.

Phase3

1. Processor p_i^w checks $token_i$. If it determines that p_i is faulty, it generates $test_{p_i^w}(p_i, 1)$. Otherwise, it generates $test_{p_i^w}(p_i, 0)$.
2. Processor p_i^s checks $token_i$. If it determines that p_i is faulty, it generates $test_{p_i^s}(p_i, 1)$. Otherwise, it generates $test_{p_i^s}(p_i, 0)$.
3. At the start of time $(tP + stc + ttd + 2tcc)$ processor p_i, for $0 \leq i < N$, sends the test result $test_{p_i}(p_i^e, -)$ and $test_{p_i}(p_i^n, -)$ of the processors p_i^e and p_i^n it tested to the host.
4. If the host does not receive a test message from a processor p_i by time $(tP + stc + ttd + 2tcc + 2mtd)$, it concludes that p_i is faulty.

5. If a processor p_i has been diagnosed as fault-free by both its testers, update $FV[tr_1, \ldots, tr_{i-1}, 0, tr_{i+1}, \ldots, tr_N]$. Otherwise, if it has been diagnosed as faulty by both its testers, update $FV[tr_1, \ldots, tr_{i-1}, 1, tr_{i+1}, \ldots, tr_N]$.

6. While tr_i is not set to 0/1 for some i in $FV[\,]$, do
   - If a processor p_j has been diagnosed as faulty by one of the testers and fault-free by the other one, identify the faulty and/or the fault-free processor from $FV[\,]$ (if already determined). p_j is fault-free if the fault-free (faulty) tester has diagnosed it as fault-free (faulty). Otherwise, p_j is faulty.

3 Conclusion

We proposed a periodic token-based diagnostic algorithm for real-time systems, which employs any multiprocessor scheduling policy. All the processors start executing the diagnostic algorithm at the same time. Each processor performs a test of itself and generates a token which contains the test result. The token is sent to two of its neighboring processors to determine if a failure has occurred. Unlike most of the previous works, we do not assume that a faulty processor stops functioning. In our model, a faulty processor may behave in erratic manners when checking upon a processor it is assigned to test. Our approach is suitable for those systems where processors fail frequently. We have given our results for torus interconnection networks.

References

1. A. A. Bertossi, M. Bonometto, and L. V. Mancini. Increasing Processor Utilization in Hard-Real-Time Systems with Checkpoints. *Real-Time Systems Journal*, 9(1):5–29, July 1995.
2. A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Transactions on Computers*, 44(12):1429–1442, Dec. 1995.
3. S. H. Hosseini. Fault-Tolerant Scheduling of Independent Tasks and Concurrent Fault-Diagnosis in Multiprocessor Systems. pages 343–350. Proceedings IEEE International Conference on Parallel Processing, Illinois, Aug. 1988.
4. S. H. Hosseini and N. Jamal. Efficient Distributed Algorithms for Self Testing of Multiple Processor Systems. *IEEE Transactions on Computers*, 41(1):1397–1409, Nov. 1992.
5. K. Hwang. *Advanced Computer Architecture. Parallelism, Scalability, Programmability.* McGraw Hill, 1993.
6. F. J. Meyer and D. K. Pradhan. Dynamic Testing Strategy for Distributed Systems. *IEEE Transactions on Computers*, 39(3), Mar. 1989.
7. Y. Oh and S. H. Son. Fault-Tolerant Real-Time Multiprocessor Scheduling. Technical Report TR-92-09, University of Virginia, April 1992.

8. S. Ponzio. Bounds on the Time to Detect Failures Using Bounded-Capacity Message Links. pages 236–245. Proceedings of Real-Time Systems Symposium, Phoenix, AZ, Dec. 1992.
9. K. Ramamritham and J. A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of the IEEE*, 82(1):55–67, Jan. 1994.
10. K. Ramamritham, J. A. Stankovic, and P. F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.
11. J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.
12. J. Sun, R. Bettati, and J. W. S. Liu. An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems. pages 18–22. 11th IEEE Workshop on Real-Time Operating Systems and Software, Seattle, Wash., May 1994.
13. T. Tsuchiya, Y. Kakuda, and T. Kikuno. A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems. pages 197–202. Proceedings of the 2nd International Workshop on Computing and Applications, Tokyo, Japan, Oct. 1995.
14. C. L. Yang and G. M. Masson. An Efficient Algorithm for Multiprocessor Fault Diagnosis Using the Comparison Approach. pages 238–243. The 16th Annual International Symposium on Fault-Tolerant Computing Systems, FTCS-16, 1986.

This article was processed using the LaTeX macro package with LLNCS style